

# ECE 530: CLOUD COMPUTING

## HOMEWORK #4: AUTOMATED DEPLOYMENT WITH ANSIBLE

AMBER DISHER – 101839171 – ADISHER1@UNM.EDU

MARSHALL HUNDEMER – 101736010 – MHUNDEMER@UNM.EDU

DAVID KIRBY – 101652098 – DAVIDKIRBY@UNM.EDU

SPRING 2021



## Contents

<b>1</b>	<b>List of Figures</b>	<b>1</b>
<b>2</b>	<b>Abstract</b>	<b>2</b>
<b>3</b>	<b>Introduction</b>	<b>2</b>
<b>4</b>	<b>Deployment</b>	<b>2</b>
<b>5</b>	<b>Conclusion</b>	<b>7</b>
<b>6</b>	<b>Extra</b>	<b>7</b>
<b>7</b>	<b>Appendix</b>	<b>7</b>

## List of Figures

1	Ansible installation via Homebrew.	2
2	Ansible installation verification.	3

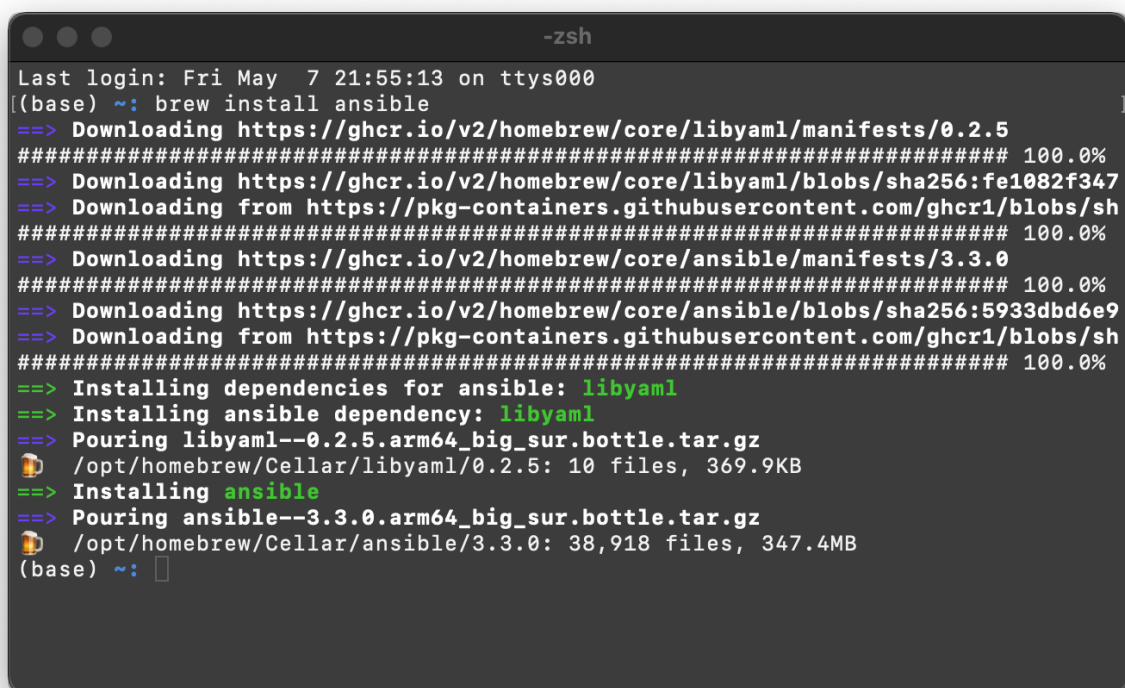
## 2 Abstract

## 3 Introduction

## 4 Deployment

### Ansible installation via Homebrew

```
brew install ansible
```



```
-zsh
Last login: Fri May 7 21:55:13 on ttys000
(base) ~: brew install ansible
==> Downloading https://ghcr.io/v2/homebrew/core/libyaml/manifests/0.2.5
##### 100.0%
==> Downloading https://ghcr.io/v2/homebrew/core/libyaml/blobs/sha256:fe1082f347
==> Downloading from https://pkg-containers.githubusercontent.com/ghcr1/blobs/sh
##### 100.0%
==> Downloading https://ghcr.io/v2/homebrew/core/ansible/manifests/3.3.0
##### 100.0%
==> Downloading https://ghcr.io/v2/homebrew/core/ansible/blobs/sha256:5933dbd6e9
==> Downloading from https://pkg-containers.githubusercontent.com/ghcr1/blobs/sh
##### 100.0%
==> Installing dependencies for ansible: libyaml
==> Installing ansible dependency: libyaml
==> Pouring libyaml--0.2.5.arm64_big_sur.bottle.tar.gz
🍺 /opt/homebrew/Cellar/libyaml/0.2.5: 10 files, 369.9KB
==> Installing ansible
==> Pouring ansible--3.3.0.arm64_big_sur.bottle.tar.gz
🍺 /opt/homebrew/Cellar/ansible/3.3.0: 38,918 files, 347.4MB
(base) ~: 
```

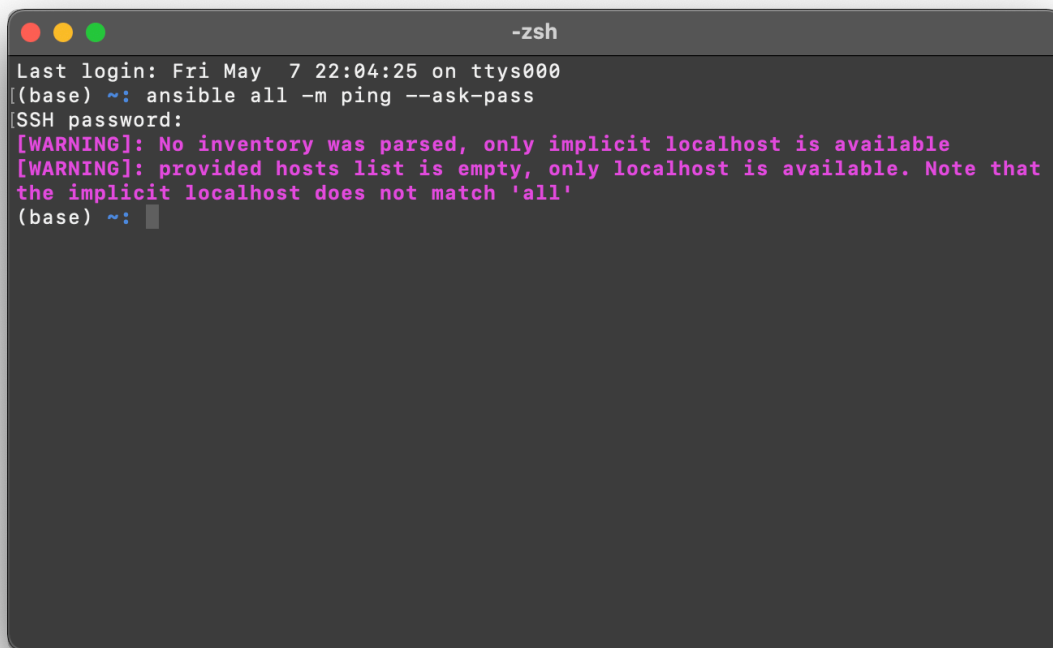
Figure 1: Ansible installation via Homebrew.

### Ansible installation verification

```
ansible all -m ping --ask-pass
```

### Pulls latest version of MongoDB from Docker Hub

```
docker pull mongo
```

A terminal window titled '-zsh' showing the output of an Ansible command. The output includes a login message, the command 'ansible all -m ping --ask-pass', a password prompt, and two warning messages in pink text. The first warning states that no inventory was parsed and only the implicit localhost is available. The second warning states that the provided hosts list is empty and only localhost is available, noting that the implicit localhost does not match 'all'. The prompt '(base) ~:' is shown at the end.

```
-zsh
Last login: Fri May  7 22:04:25 on ttys000
(base) ~: ansible all -m ping --ask-pass
SSH password:
[WARNING]: No inventory was parsed, only implicit localhost is available
[WARNING]: provided hosts list is empty, only localhost is available. Note that
the implicit localhost does not match 'all'
(base) ~: 
```

Figure 2: Ansible installation verification.

Once Docker was installed we needed to set up a network to which our databases could join. Docker has default networks installed:

- host – For standalone containers, removes network isolation between the container and the Docker host, and uses the host's networking directly.
- bridge – The default network driver. Bridge networks are usually used when your applications run in standalone containers that need to communicate.

The default bridge network allow us to configure the network, but all the containers use the same settings, such as MTU and iptables rules. In addition, configuring the default bridge network happens outside of Docker itself, and requires a restart of Docker. Creating our own bridge network, created and configured using `docker network create`, allows different groups of applications to have different network requirements, allows us to configure each user-defined bridge separately. Containers connected to the same user-defined bridge network effectively expose all ports to each other.

List all networks created in Docker

```
docker network ls
```

Create our user-defined network

```
docker network create mongo-net
```

Now that our network was created (see , we began creating our containers from the Mongo image and connecting them altogether.

#### Create first MongoDB image

```
docker run -p 30001:27017 --name mongo1 --net mongo-net mongo \
mongod --replSet mongo-set
```

- `docker run` – Start a container from an image
- `-p 30001:27017` – Expose port 27017 in our container, as port 30001 on the localhost
- `--name mongo1` – Name this container “mongo1”
- `--net mongo-net` – Add this container to the “mongo-net” network.
- `mongo` – the name of the image we are using to spawn this container
- `mongod --replSet mongo-set` – Run mongod while adding this mongod instance to the replica set named “mongo-set”

We can see in that we are running Ubuntu 18.04.5 LTS and MongoDB 4.4.5 on aarch64 (ARM) architecture.

We then created the secondary images, each of which needed to be run in a separate terminal tab, and finally after all database containers were created, we turned them into a replica set.

#### Create secondary MongoDB images

```
docker run -p 30002:27017 --name mongo2 --net mongo-net mongo \
mongod --replSet mongo-set
```

```
docker run -p 30003:27017 --name mongo3 --net mongo-net mongo \
mongod --replSet mongo-set
```

We started our Docker container directly into our MongoDB and began configuring the connections.

#### Connect to mongo1 and configure it to be the primary

```
docker exec -it mongo1 mongo

> db = (new Mongo('localhost:27017')).getDB('test')

> config = {
  "_id" : "mongo-set",
  "members" : [
    {
      "_id" : 0,
      "host" : "mongo1:27017"
    },
    {
      "_id" : 1,
      "host" : "mongo2:27017"
    },
    {
      "_id" : 2,
      "host" : "mongo3:27017"
    }
  ]
}
```

The results of the above commands are shown in. Then we initiated the configuration file we just created and received confirmation by noticing the change in command prompt.

#### Initiate our replica set using our just created config file

```
> rs.initiate(config)

mongo-set:PRIMARY> # Confirms that we are on PRIMARY DB
```

We then wrote data to our primary DB to verify it (see , writing a document "ECE530". It is important to note that data can only be written to the primary DB, but, as we will show later, can be read from any of the databases. If for some reason the primary database goes down, one of the others will become primary and allow for robustness.

#### Write data to mongo1 – our primary DB and then read it

```
> db.mycollection.insert({name : 'ECE530'})

> db.mycollection.find()
```

We then connect to each of our secondary databases and test to see if our document gets replicated there as well.

Test that data is being replicated to mongo2 and mongo3 respectively

```
> db2 = (new Mongo('mongo2:27017')).getDB('test')
> db2.setSecondaryOk()
> db2.mycollection.find()

> db3 = (new Mongo('mongo3:27017')).getDB('test')
> db3.setSecondaryOk()
> db3.mycollection.find()
```

## 5 Conclusion

In conclusion, we were successful in creating a Dockerfile to create images of MongoDB, although we found Dockerfiles limited. We set up a user-defined network for our databases, created three MongoDB databases from our Docker image, and designed those databases to be a replica set. We tested writing data to our primary database and confirmed that the data is indeed distributed. For future work, we would like to explore docker-compose to attempt automating this entire process.

## 6 Extra

## 7 Appendix

Dockerfile

```
# Pulls latest version of MongoDB from Docker Hub
FROM mongo
LABEL maintainer=davidkirby@unm.edu

# Automatically update Ubuntu
RUN apt-get update && apt-get install -y

# Open Port for MongoDB to connect to host
EXPOSE 27017
```